



<https://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *iteration*



<https://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

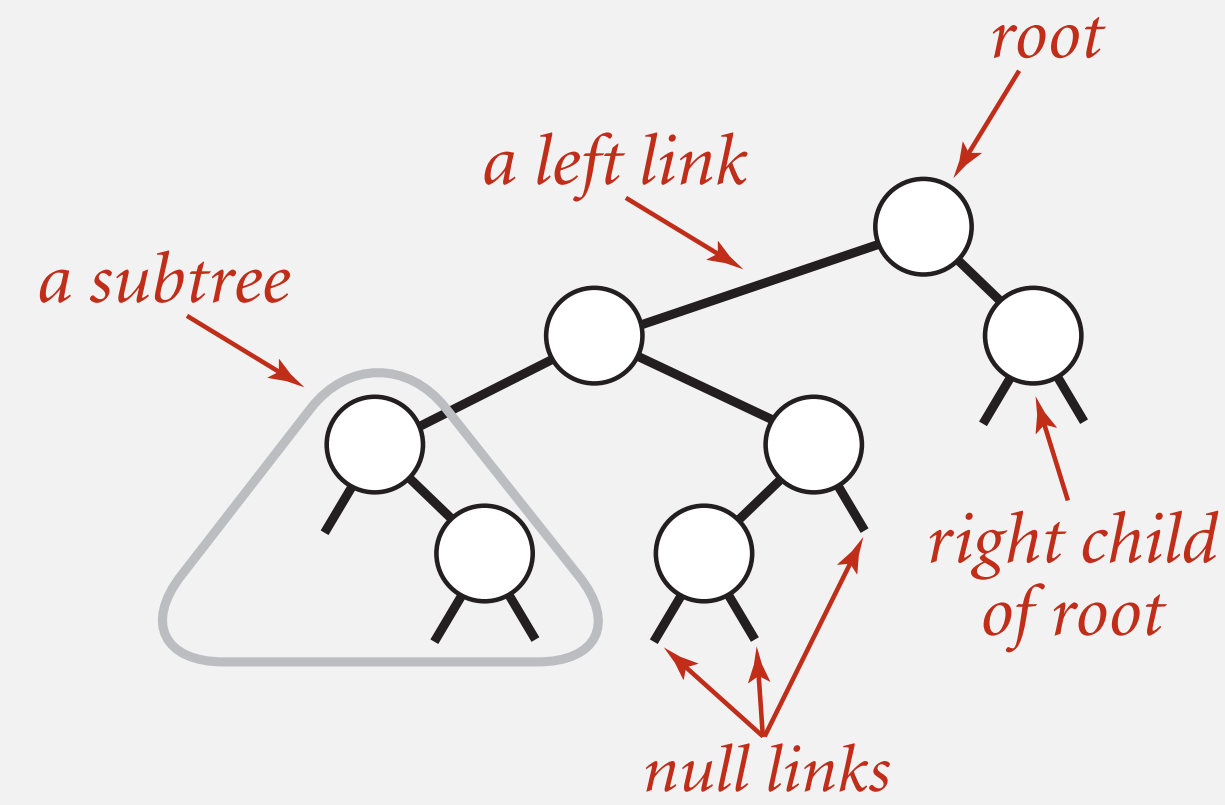
- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *iteration*

Binary search trees

Definition. A BST is a **binary tree** in **symmetric order**.

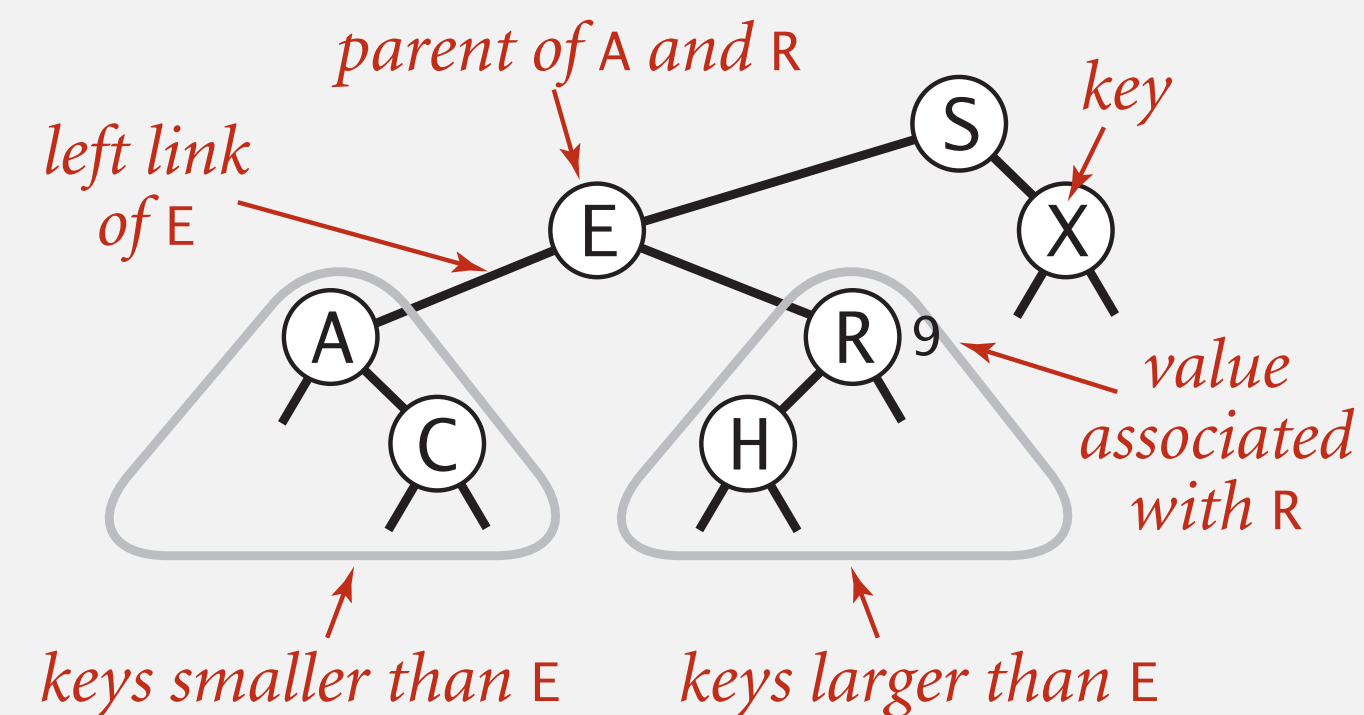
A binary tree is either:

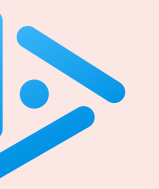
- Empty.
- A node with links to two disjoint binary trees—the left subtree and the right subtree.



Symmetric order. Each node has a key; a node's key is:

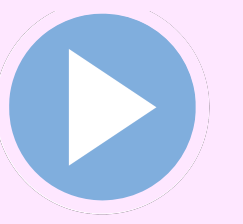
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.
- [Duplicate keys not permitted.]





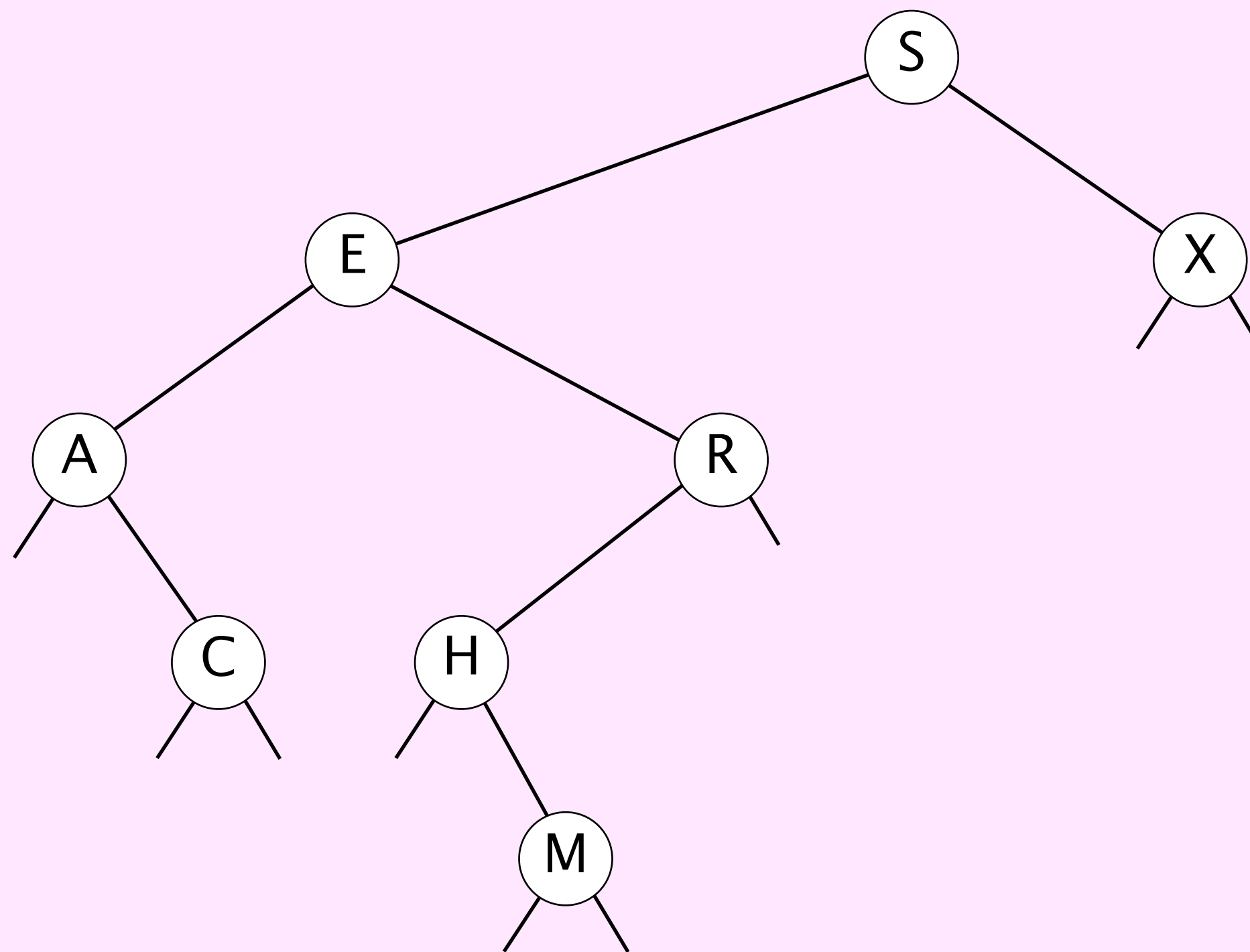
Which of the following properties hold?

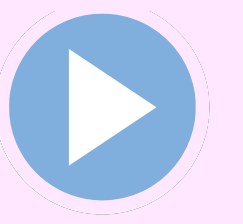
- A. If a binary tree is heap ordered, then it is symmetrically ordered.
- B. If a binary tree is symmetrically ordered, then it is heap ordered.
- C. Both A and B.
- D. Neither A nor B.



Search. If less, go left; if greater, go right; if equal, search hit.

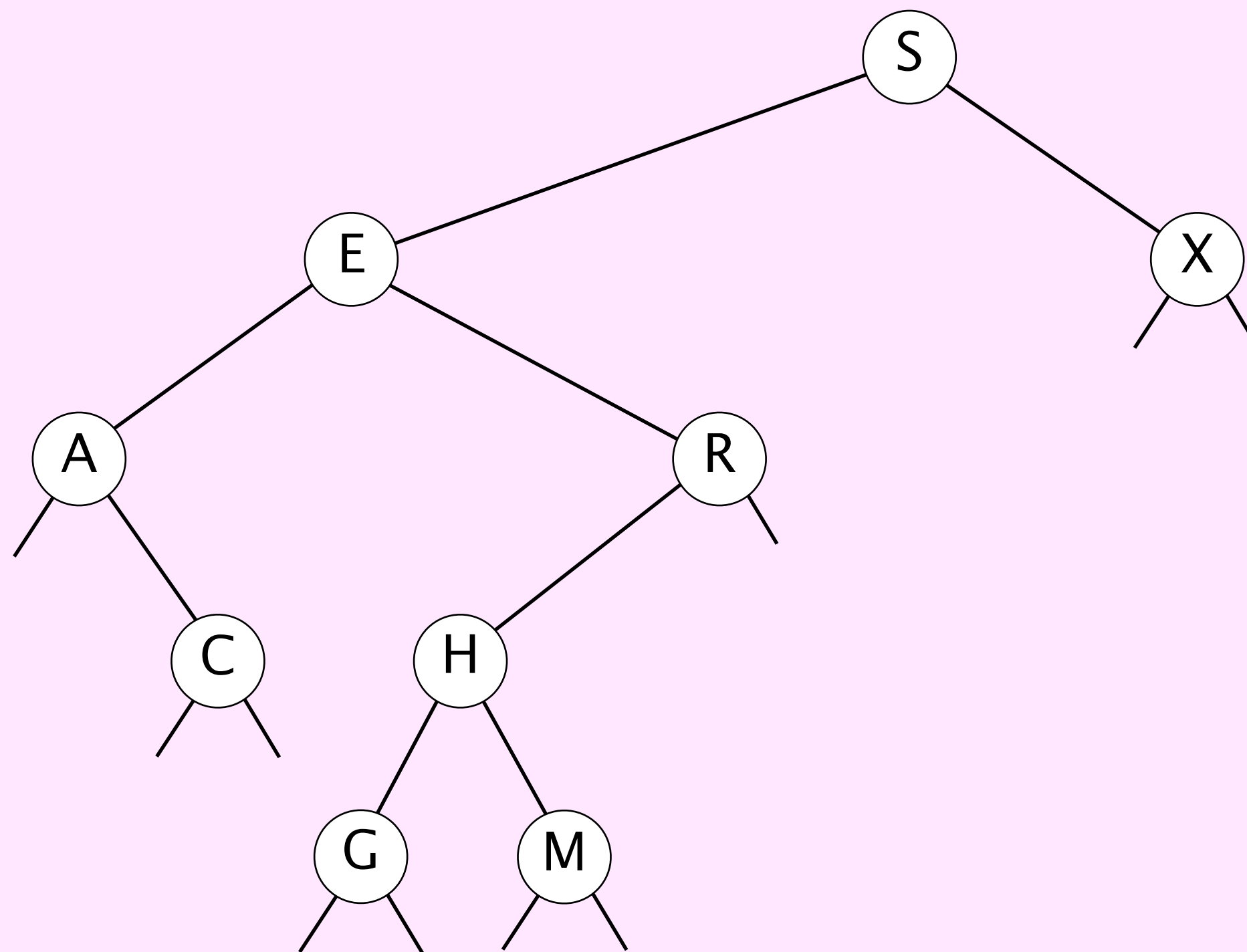
successful search for H





Insert. If less, go left; if greater, go right; if `null`, insert.

insert G



BST representation in Java

Java definition. A BST is a reference to a root **Node**.

A **Node** is composed of four fields:

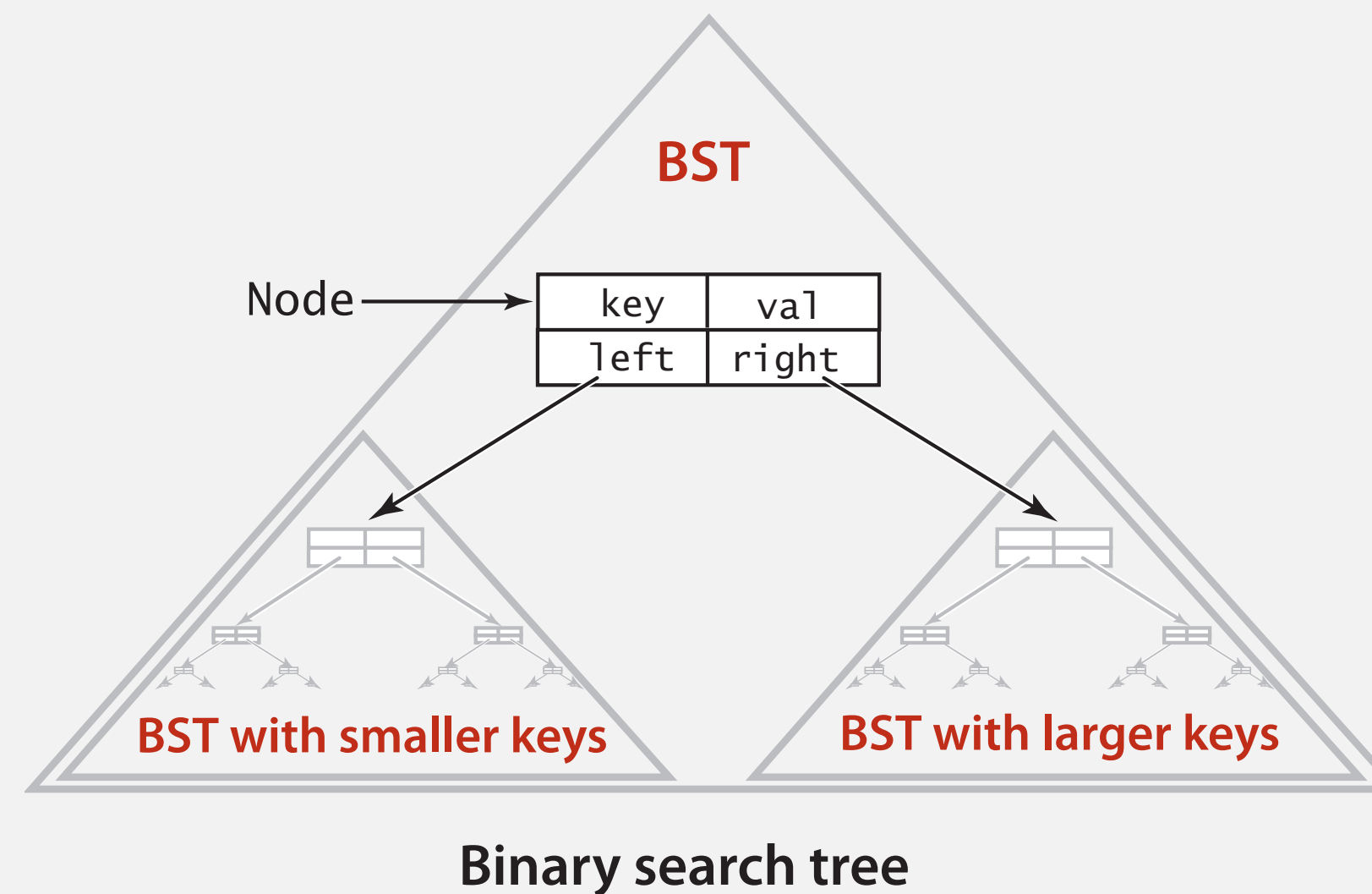
- A **Key** and a **Value**.
- A reference to the left and right subtree.

smaller keys larger keys

```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;

    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

Key and Value are generic types; Key is Comparable



BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root; ← root of BST

    private class Node
    { /* see previous slide */ }

    public void put(Key key, Value val)
    { /* see slide in this section */ }

    public Value get(Key key)
    { /* see next slide */ }

    public Iterable<Key> keys()
    { /* see slides in next section */ }

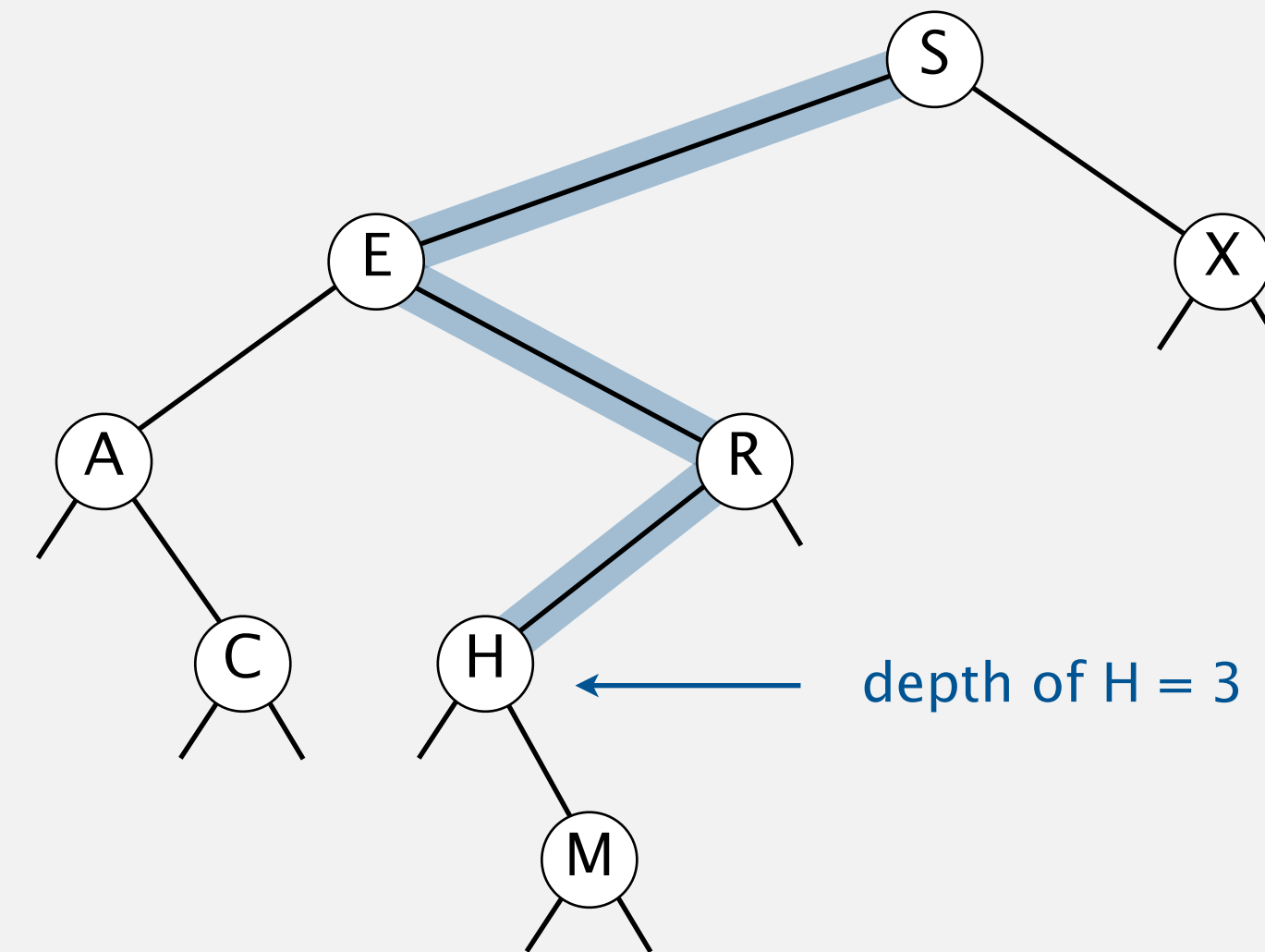
    public void delete(Key key)
    { /* see textbook */ }

}
```


BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else return x.val;
    }
    return null;
}
```



Cost. Number of compares = 1 + depth of node.

BST insert

Put. Associate value with key.

- Search for key in BST.
- Case 1: Key in BST \Rightarrow reset value.
- Case 2: Key not in BST \Rightarrow add new node.

```
public void put(Key key, Value val)
{ root = put(root, key, val); }

private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);

    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;

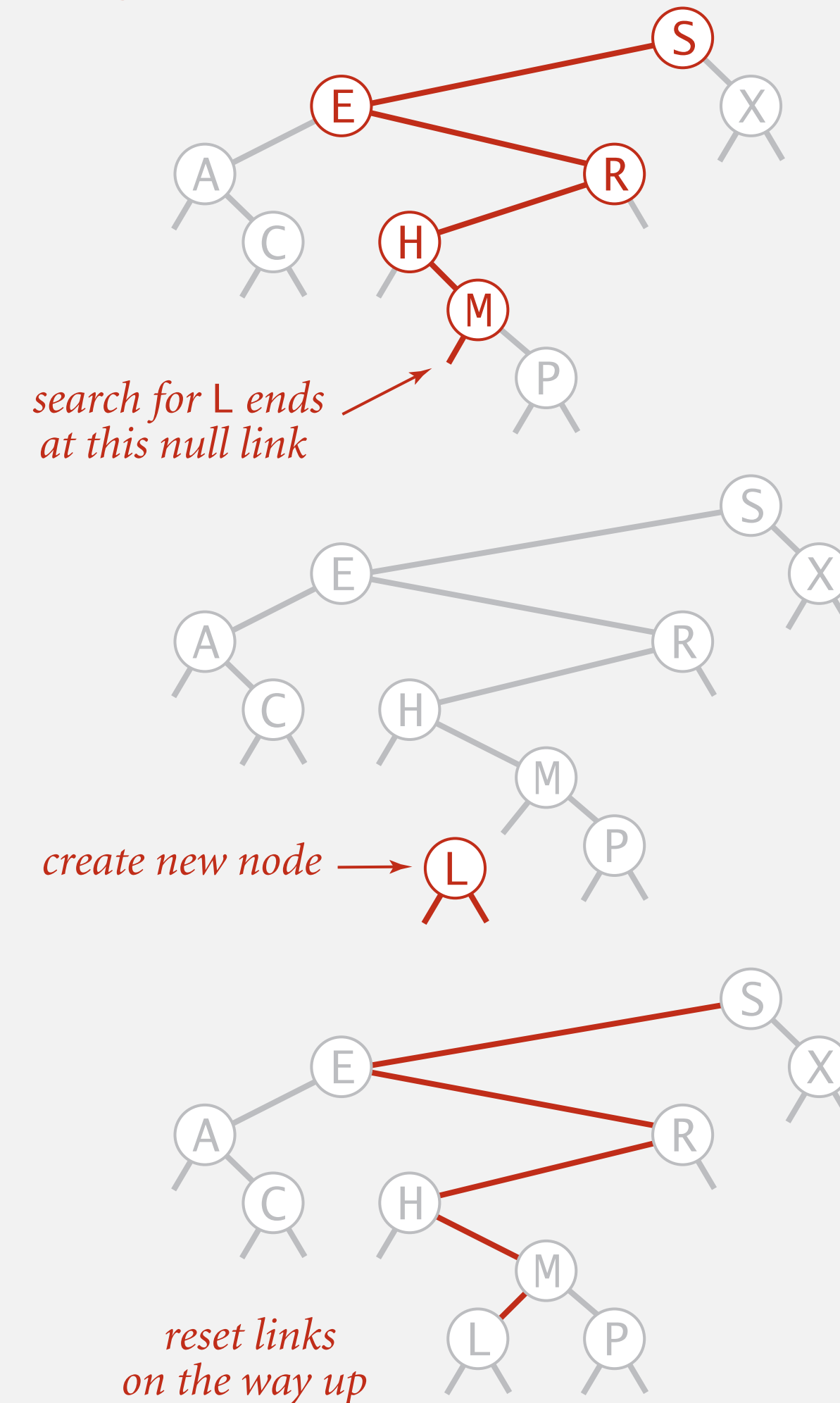
    return x;
}
```



Warning: concise but tricky code; read carefully!

Cost. Number of compares = 1 + depth of node.

inserting L

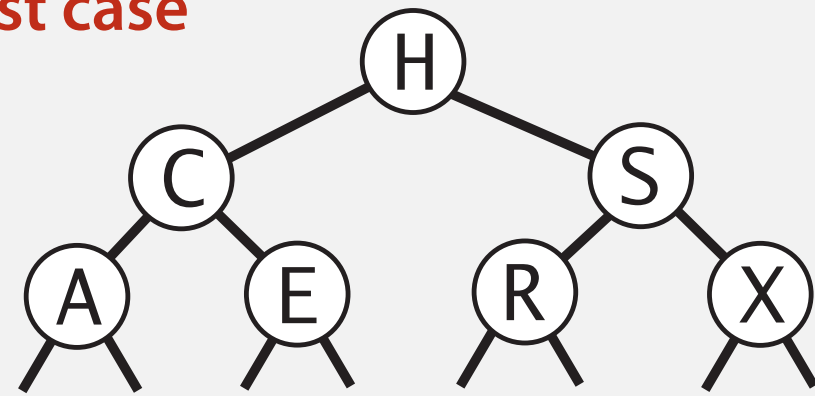


Insertion into a BST

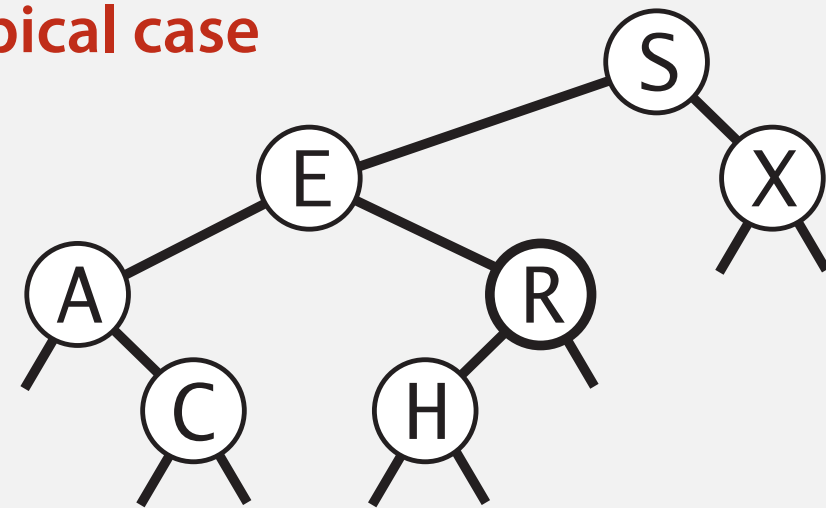
Tree shape

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.

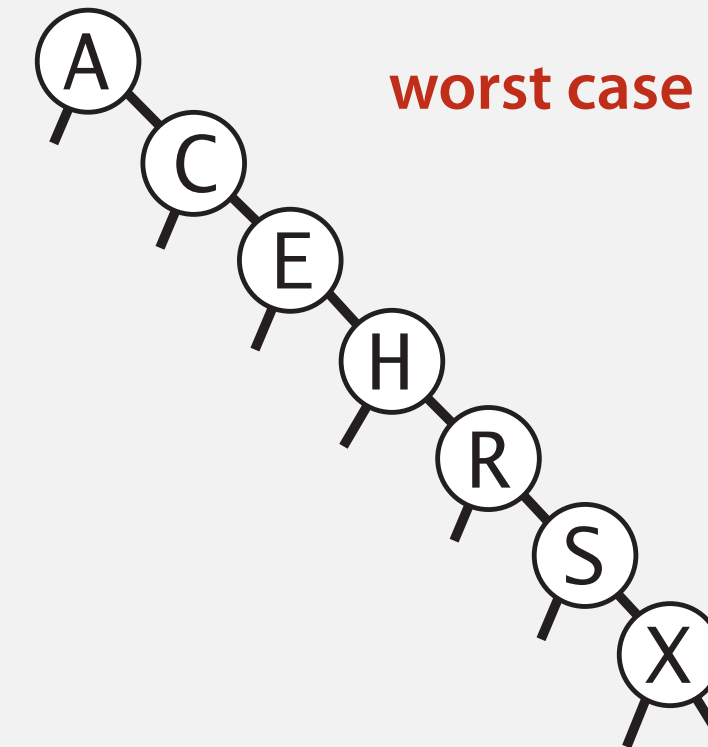
best case



typical case



worst case

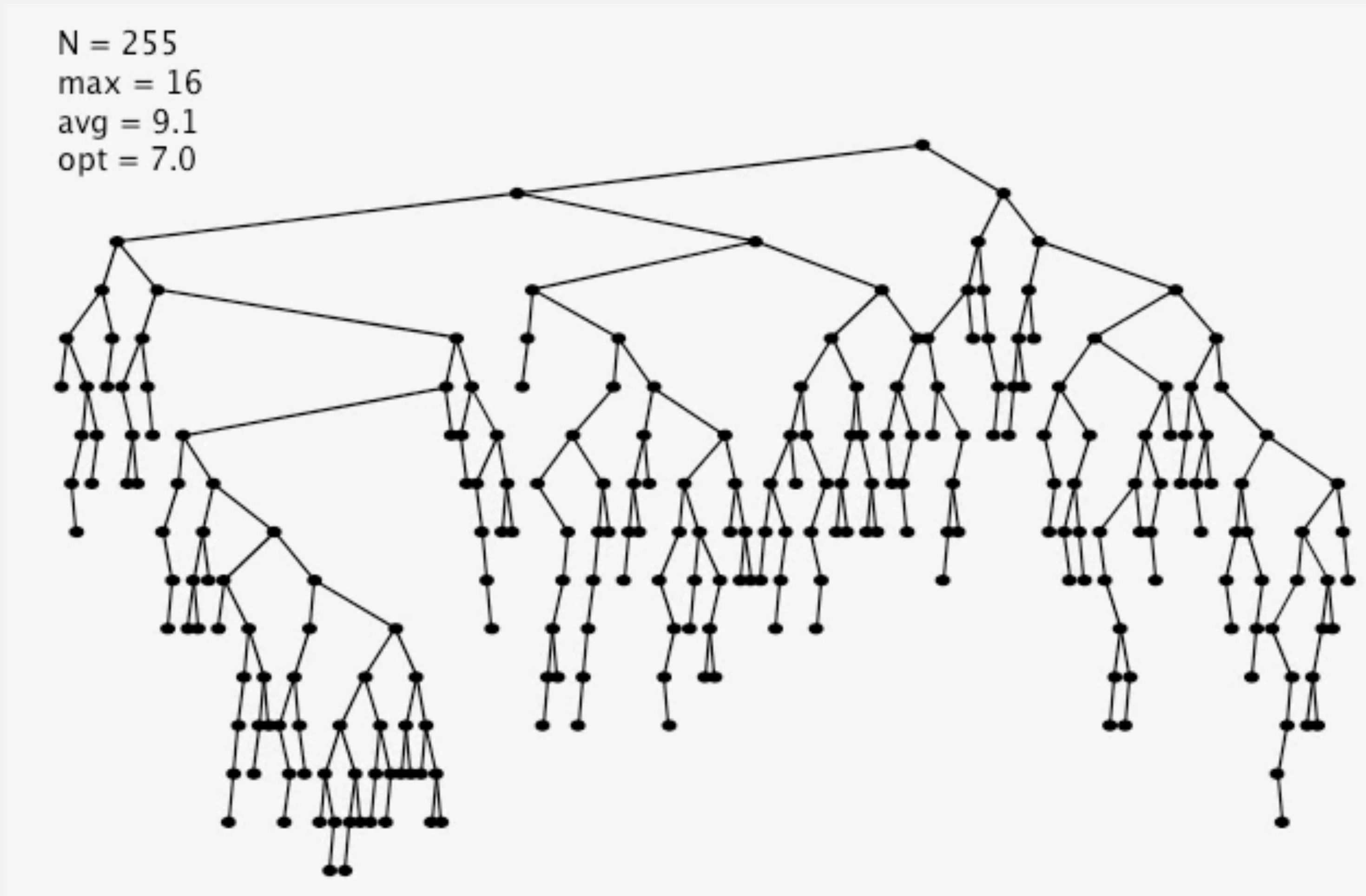


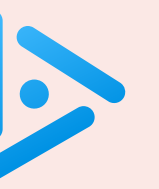
height between $\log_2 n$ and $n - 1$

Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

Ex. Insert keys in random order.

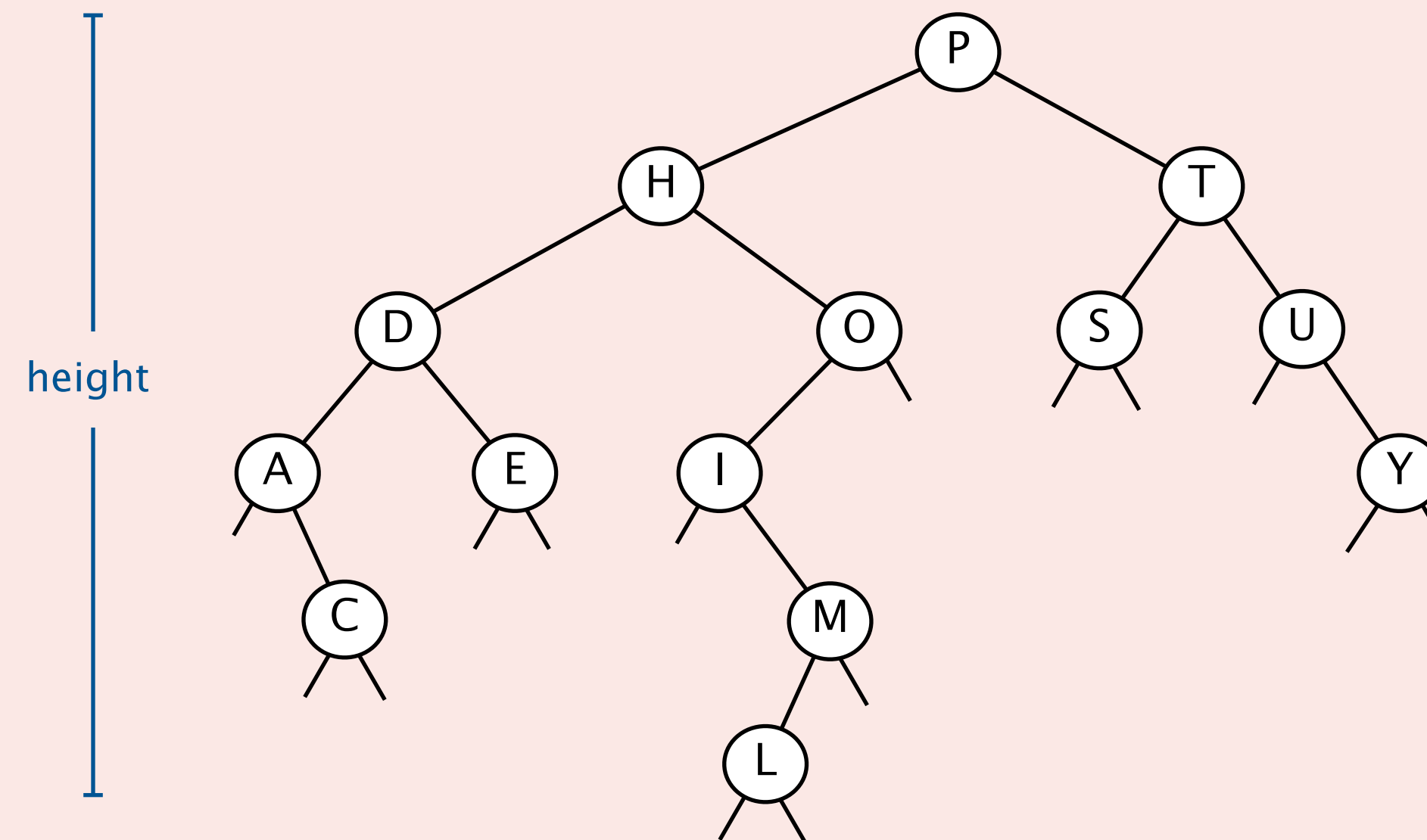




Suppose that you insert n keys in random order into a BST.

What is the **expected height** of the resulting BST?

- A. $\sim \log_2 n$
- B. $\sim 2 \ln n$
- C. $\sim 4.31107 \ln n$
- D. $\Theta(n)$



ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	n	n	n	n	<code>equals()</code>
binary search (ordered array)	$\log n$	n	$\log n$	n	<code>compareTo()</code>
BST	n	n	$\log n$	$\log n$	<code>compareTo()</code>

Why not shuffle to ensure
a (probabilistic) guarantee
of $\Theta(\log n)$ time?



<https://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

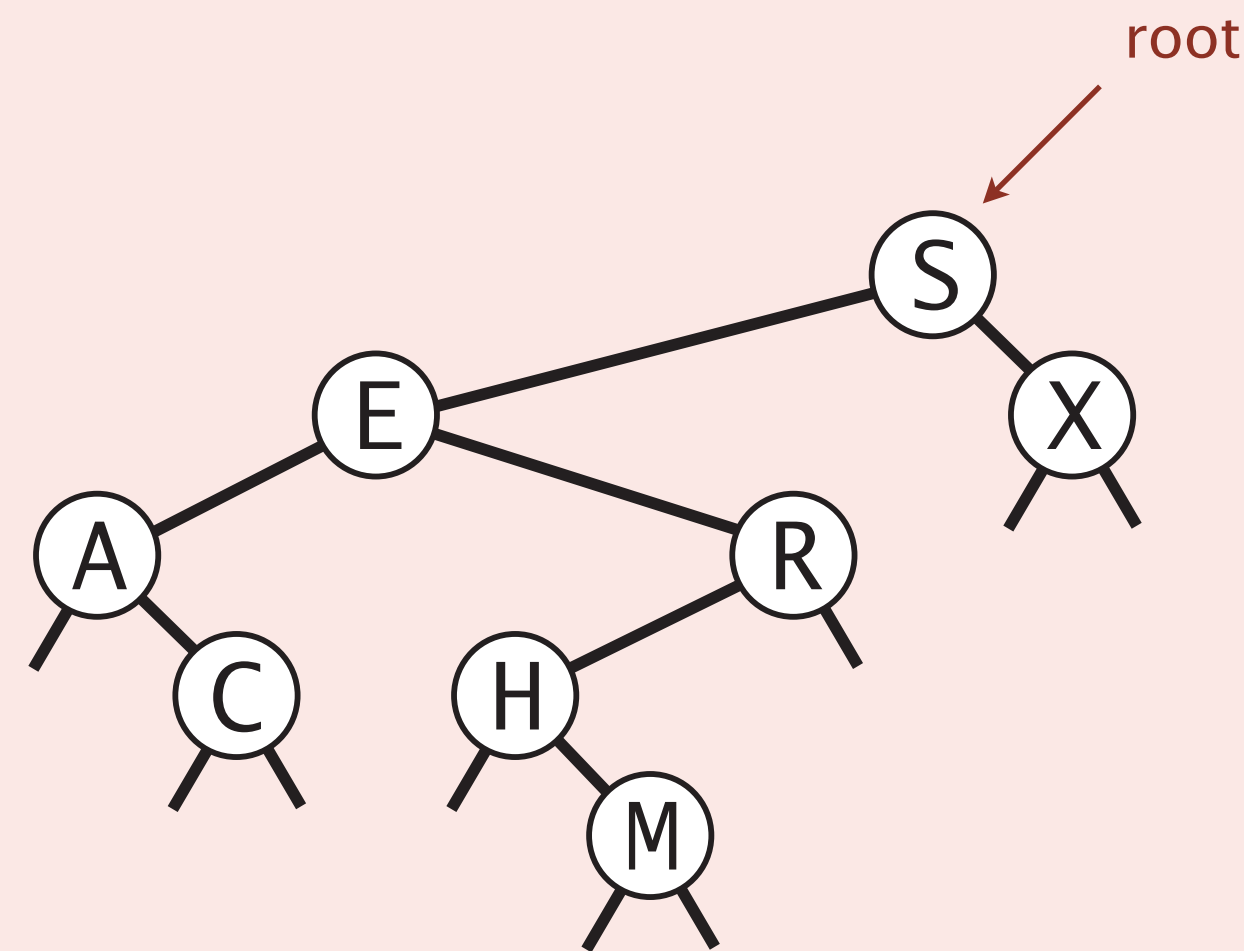
- ▶ *BSTs*
- ▶ *iteration*
- ▶ *ordered operations*



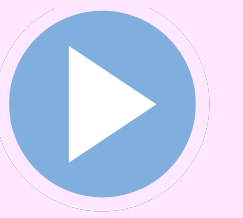
In which order does `traverse(root)` print the keys in the BST?

```
private void traverse(Node x)
{
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key);
    traverse(x.right);
}
```

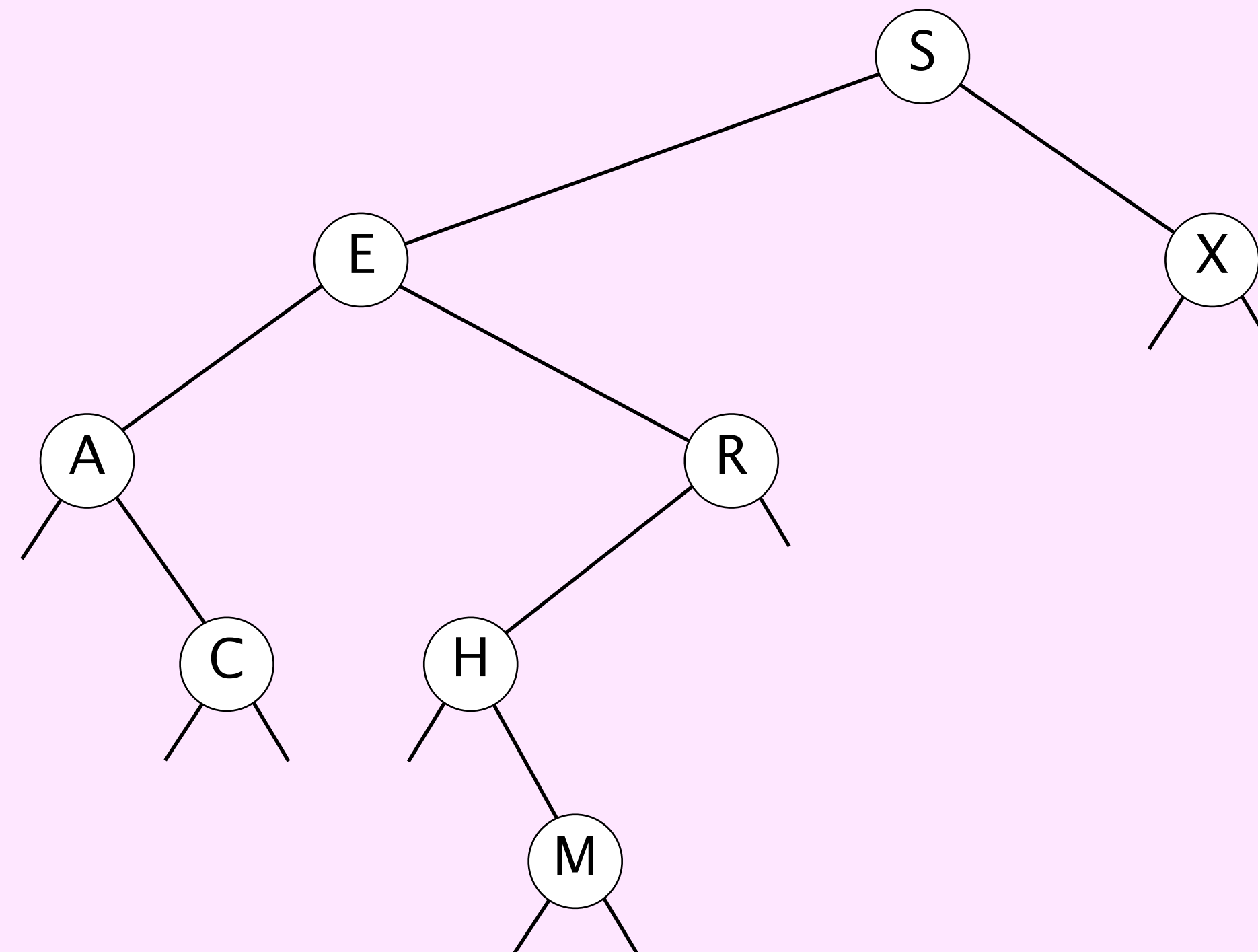
- A. A C E H M R S X
- B. S E A C R H M X
- C. C A M H R E X S
- D. S E X A R C H M



Inorder traversal



```
inorder(S)
  inorder(E)
    inorder(A)
      print A
      inorder(C)
        print C
        done C
      done A
    print E
    inorder(R)
      inorder(H)
        print H
        inorder(M)
          print M
          done M
        done H
      print R
      done R
    done E
  print S
  inorder(X)
    print X
    done X
  done S
```



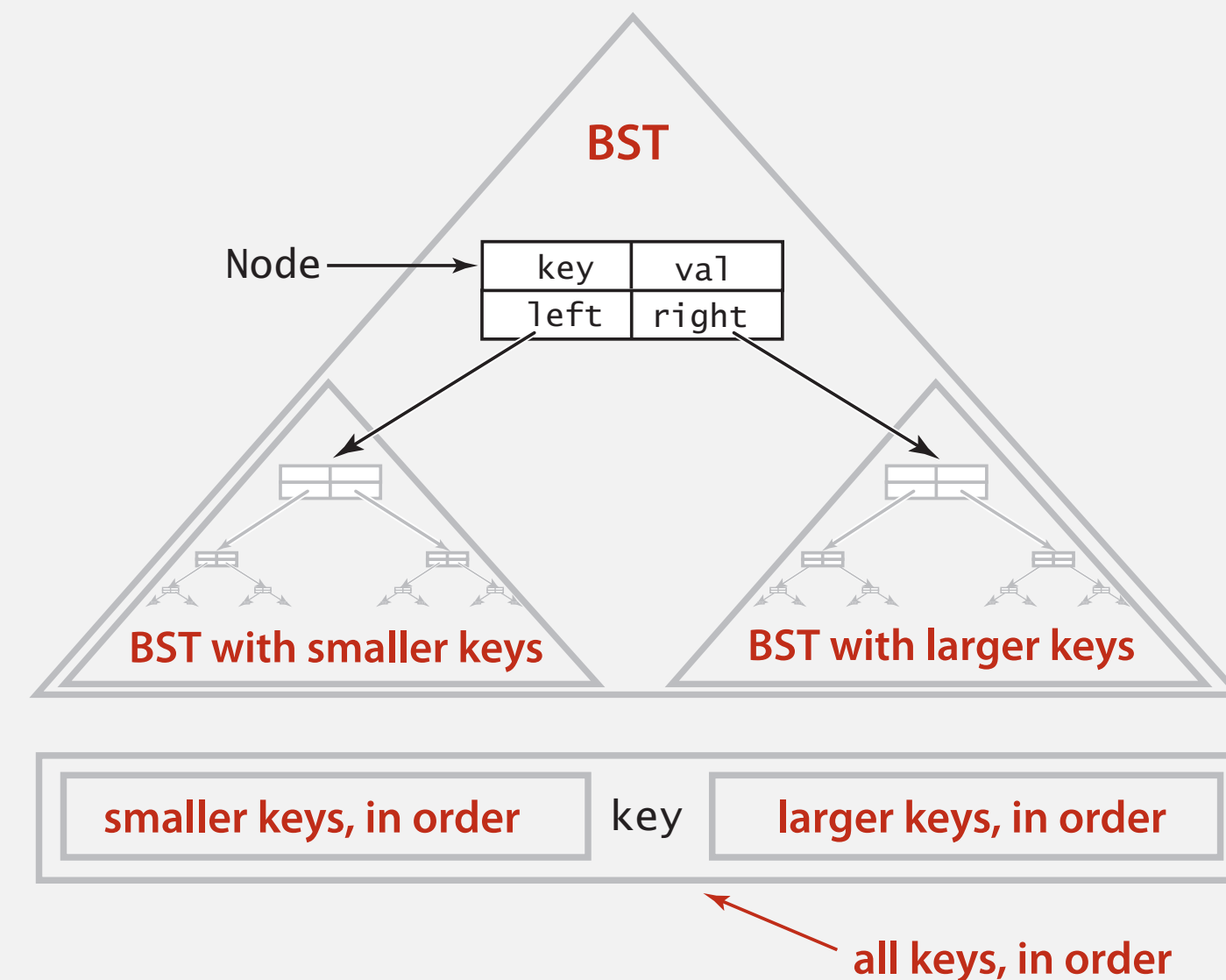
output: **A C E H M R S X**

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

add items to a collection that is Iterable
and return that collection

```
public Iterable<Key> keys()  
{  
    Queue<Key> queue = new Queue<Key>();  
    inorder(root, queue);  
    return queue;  
}  
  
private void inorder(Node x, Queue<Key> queue)  
{  
    if (x == null) return;  
    inorder(x.left, queue);  
    queue.enqueue(x.key);  
    inorder(x.right, queue);  
}
```



Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal: running time



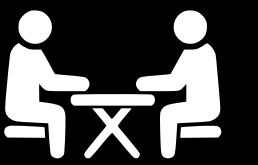
Property. Inorder traversal of a binary tree with n nodes takes $\Theta(n)$ time.

Pf. $\Theta(1)$ time per node in BST.



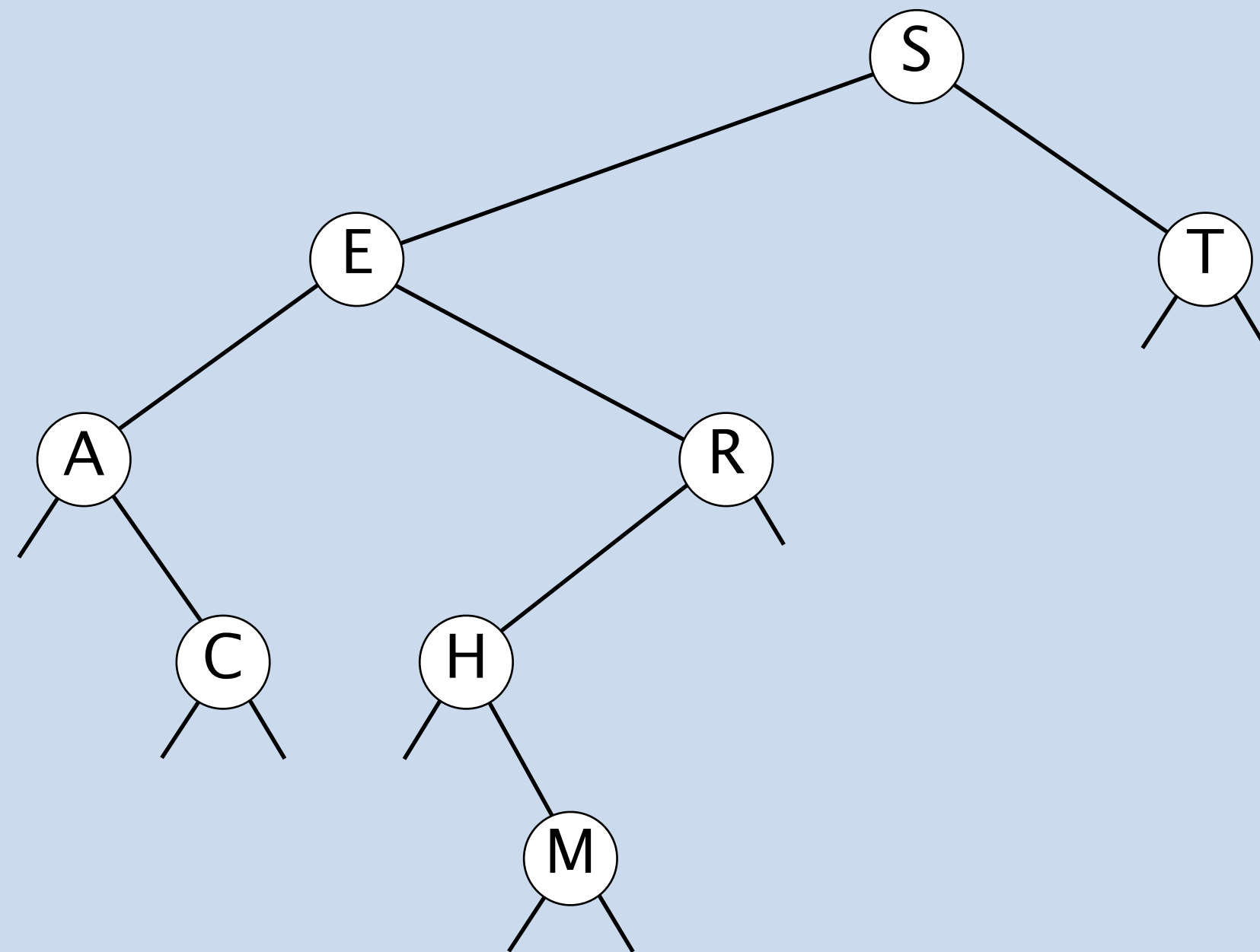
Silicon Valley (“The Blood Boy”)

LEVEL-ORDER TRAVERSAL



Level-order traversal of a binary tree.

- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...

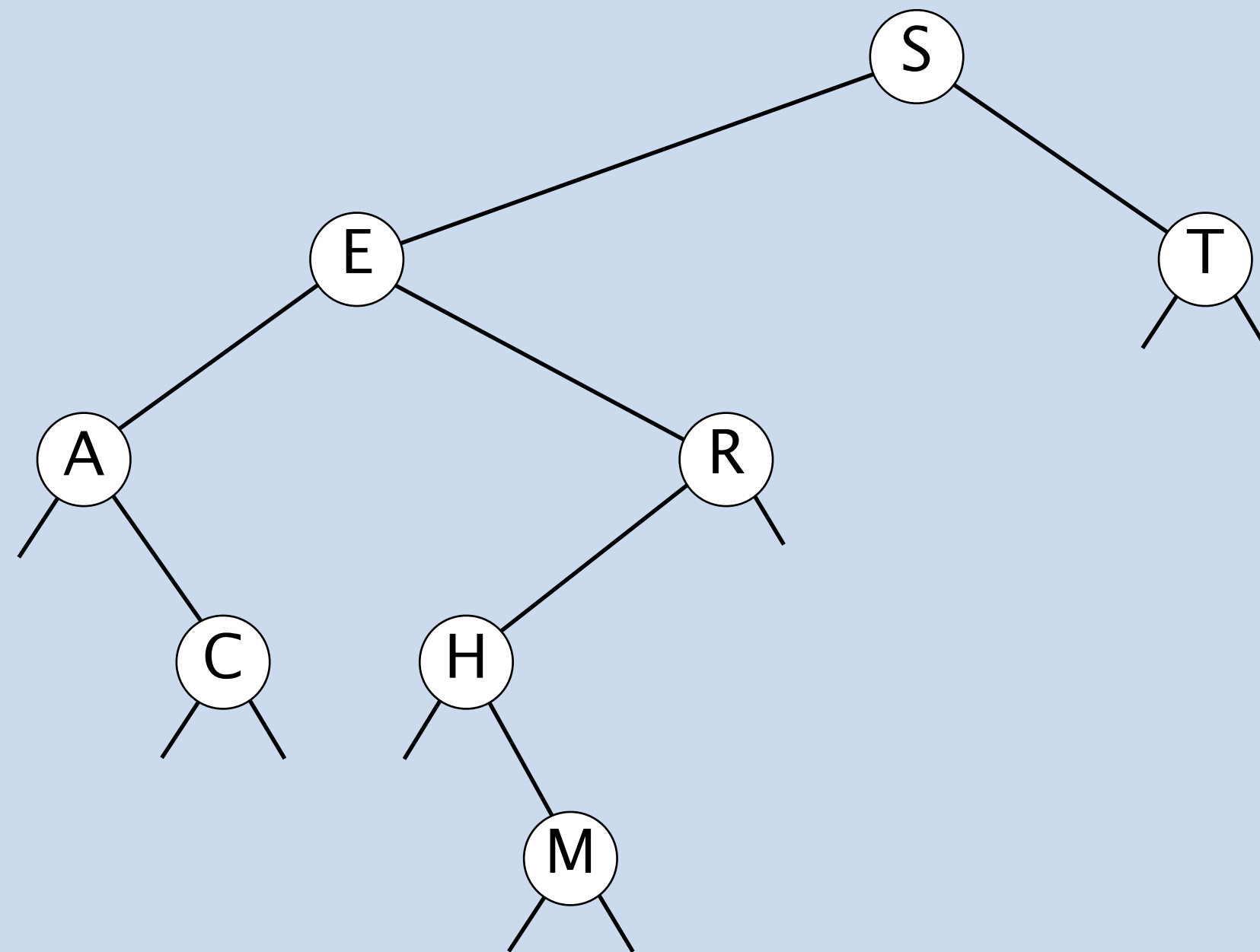


level-order traversal: **S E T A R C H M**

LEVEL-ORDER TRAVERSAL



Q1. How to compute level-order traversal of a binary tree in $\Theta(n)$ time?



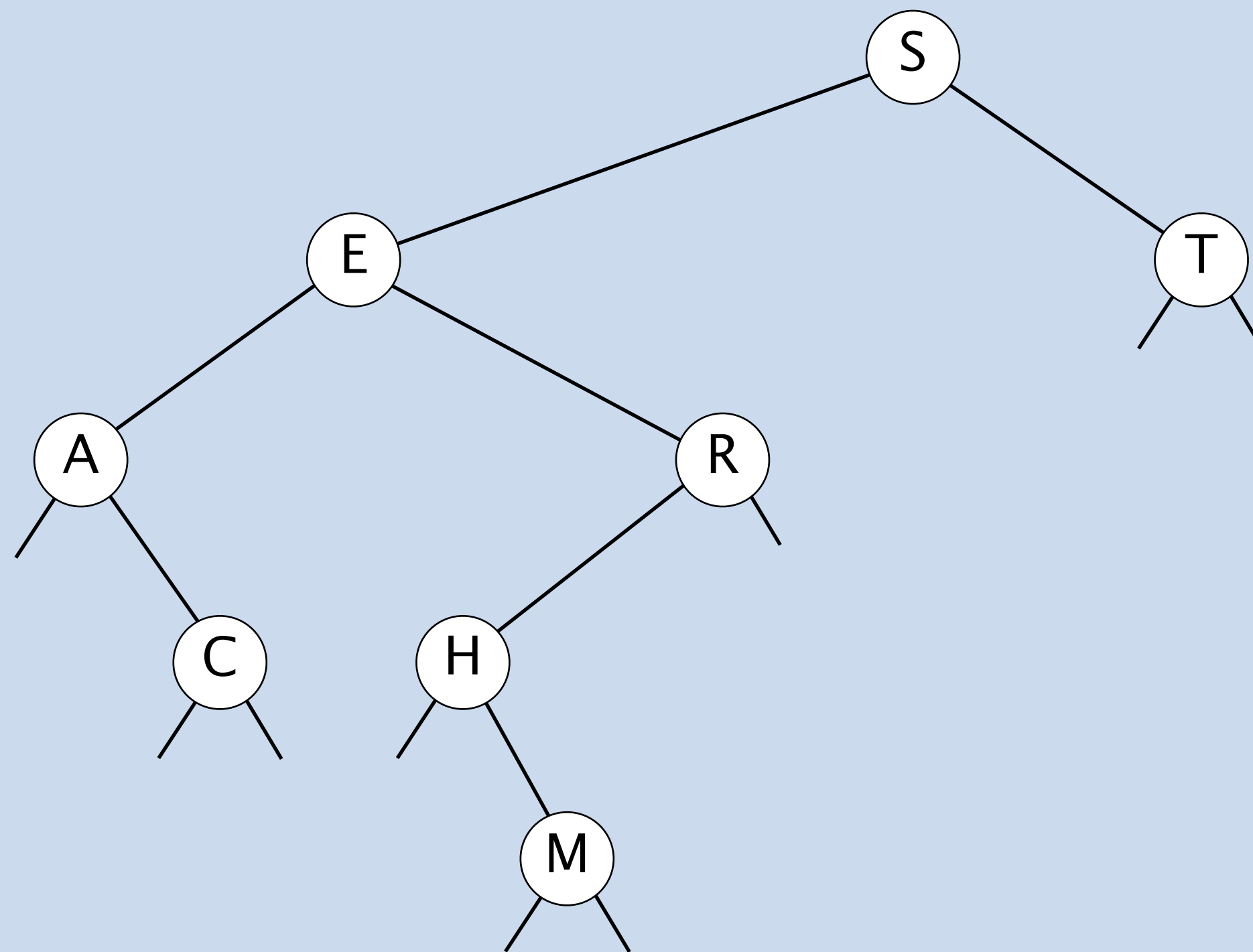
level-order traversal: **S E T A R C H M**



Q2. Given the level-order traversal of a BST, how to (uniquely) reconstruct?

Ex. ~~S~~ ~~E~~ ~~T~~ ~~A~~ ~~R~~ ~~C~~ ~~H~~ ~~M~~

↑
needed for Quizzera quizzes





<https://algs4.cs.princeton.edu>

3.2 BINARY SEARCH TREES

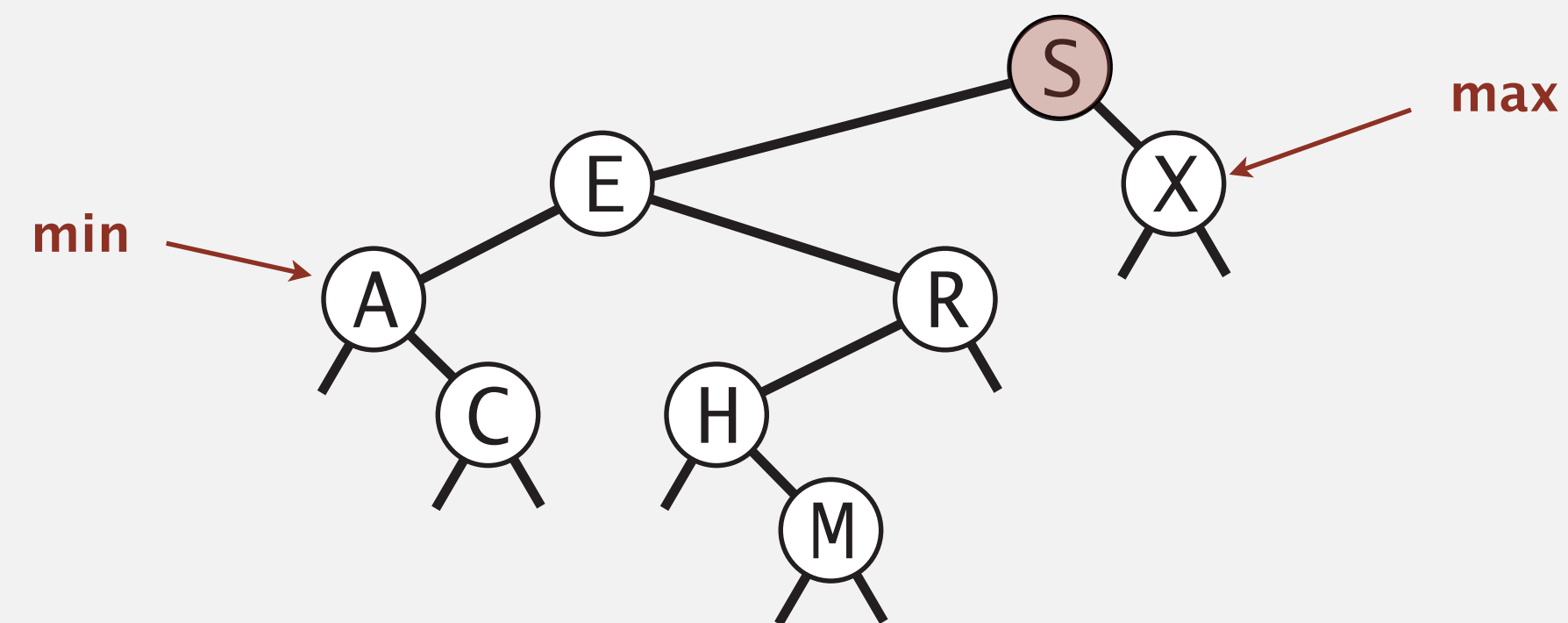
- ▶ *BSTs*
- ▶ *iteration*
- ▶ *ordered operations*

Minimum and maximum

Minimum. Smallest key in BST.

Maximum. Largest key in BST.

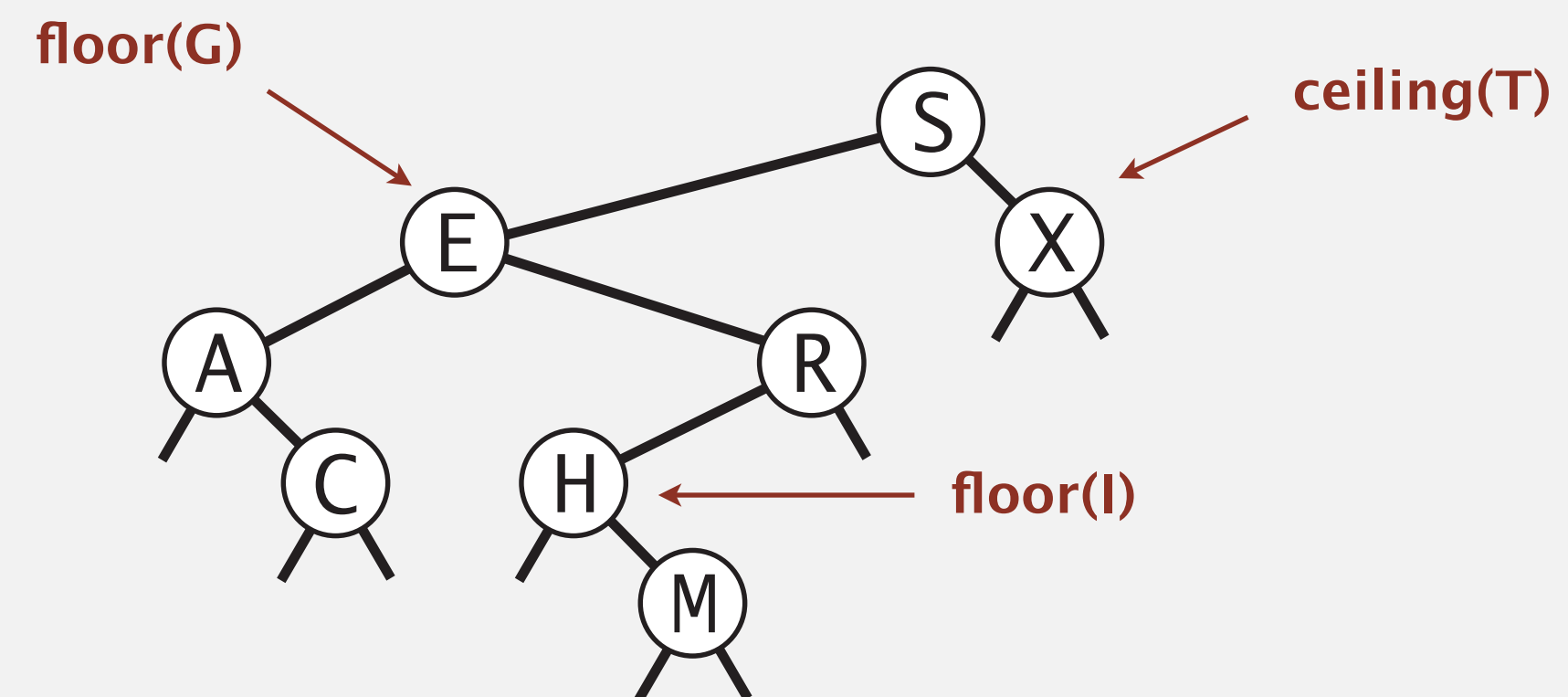
Q. How to find the min / max?



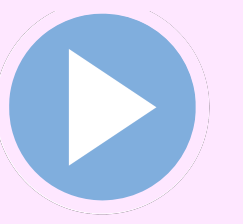
Floor and ceiling

Floor. Largest key in BST \leq query key.

Ceiling. Smallest key in BST \geq query key.



Computing the floor

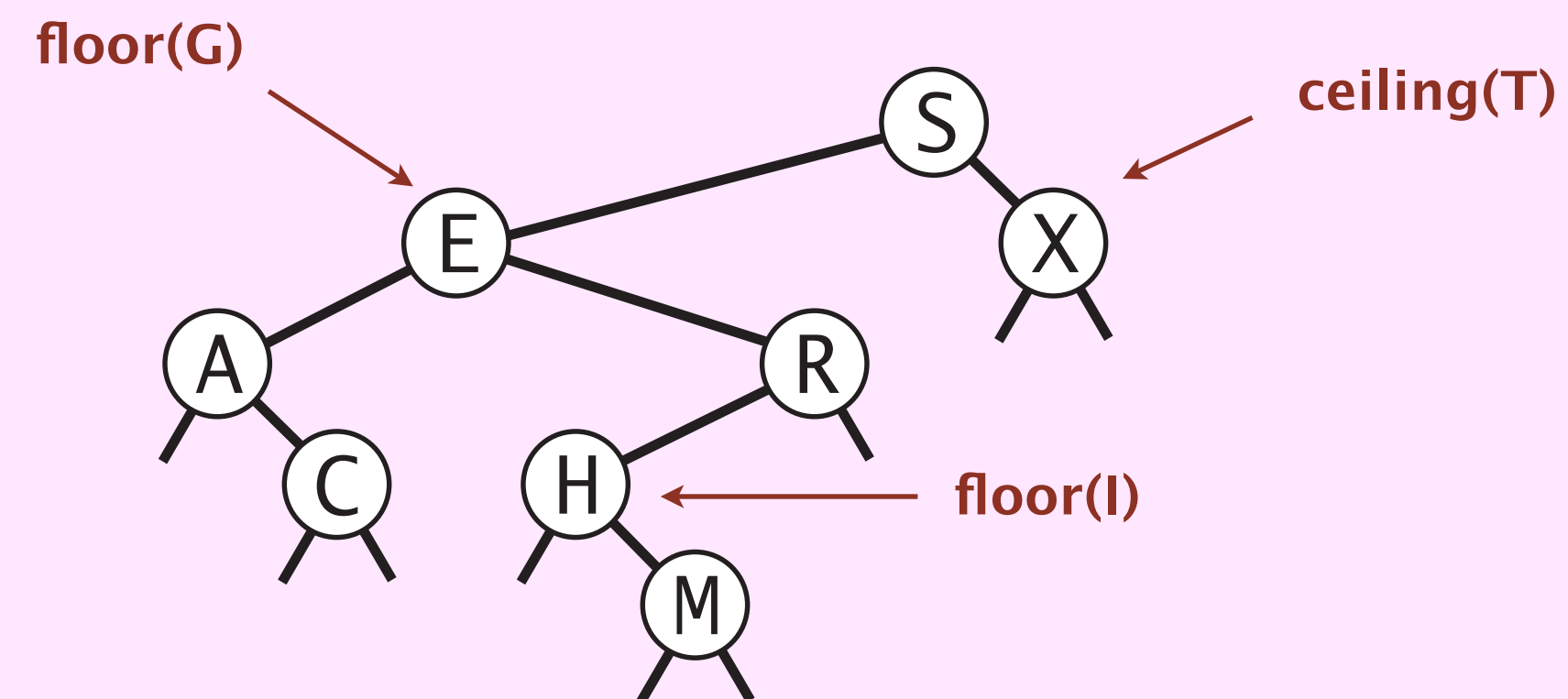


Floor. Largest key in BST \leq query key.

Ceiling. Smallest key in BST \geq query key.

Key idea.

- To compute `floor(key)` or `ceiling(key)`, search for `key`.
- Both `floor(key)` and `ceiling(key)` are on search path.
- Moreover, as you go down search path, any candidates get better and better.



Computing the floor: Java implementation

Invariant 1. The floor is either **champ** or in subtree rooted at **x**.

Invariant 2. Node **x** is in the right subtree of node containing **champ**. ← assuming champ is not null

```
public Key floor(Key key)
{ return floor(root, key, null); }
```

```
private Key floor(Node x, Key key, Key champ)
{
```

```
    if (x == null) return champ;
```

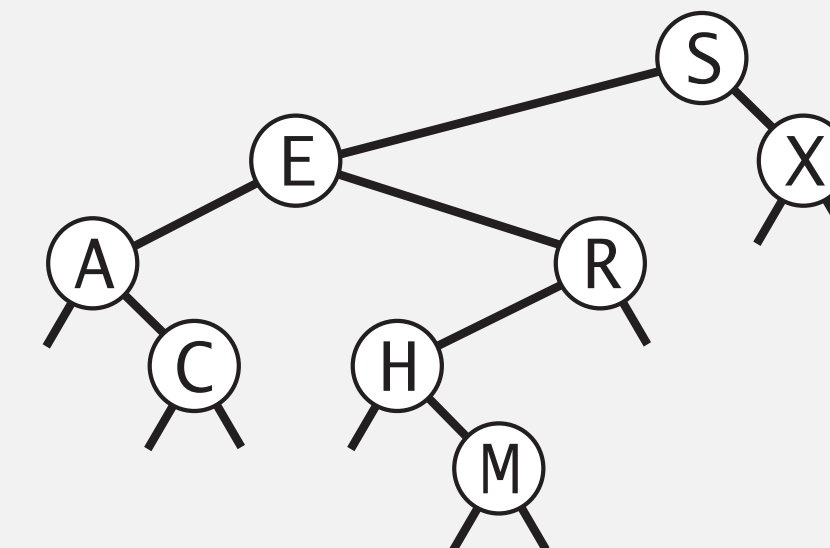
```
    int cmp = key.compareTo(x.key);
```

```
    if (cmp < 0) return floor(x.left, key, champ);
```

```
    else if (cmp > 0) return floor(x.right, key, x.key);
```

```
    else return x.key;
```

```
}
```



champ must be floor

key in node x is too large
(floor can't be in right subtree of x)

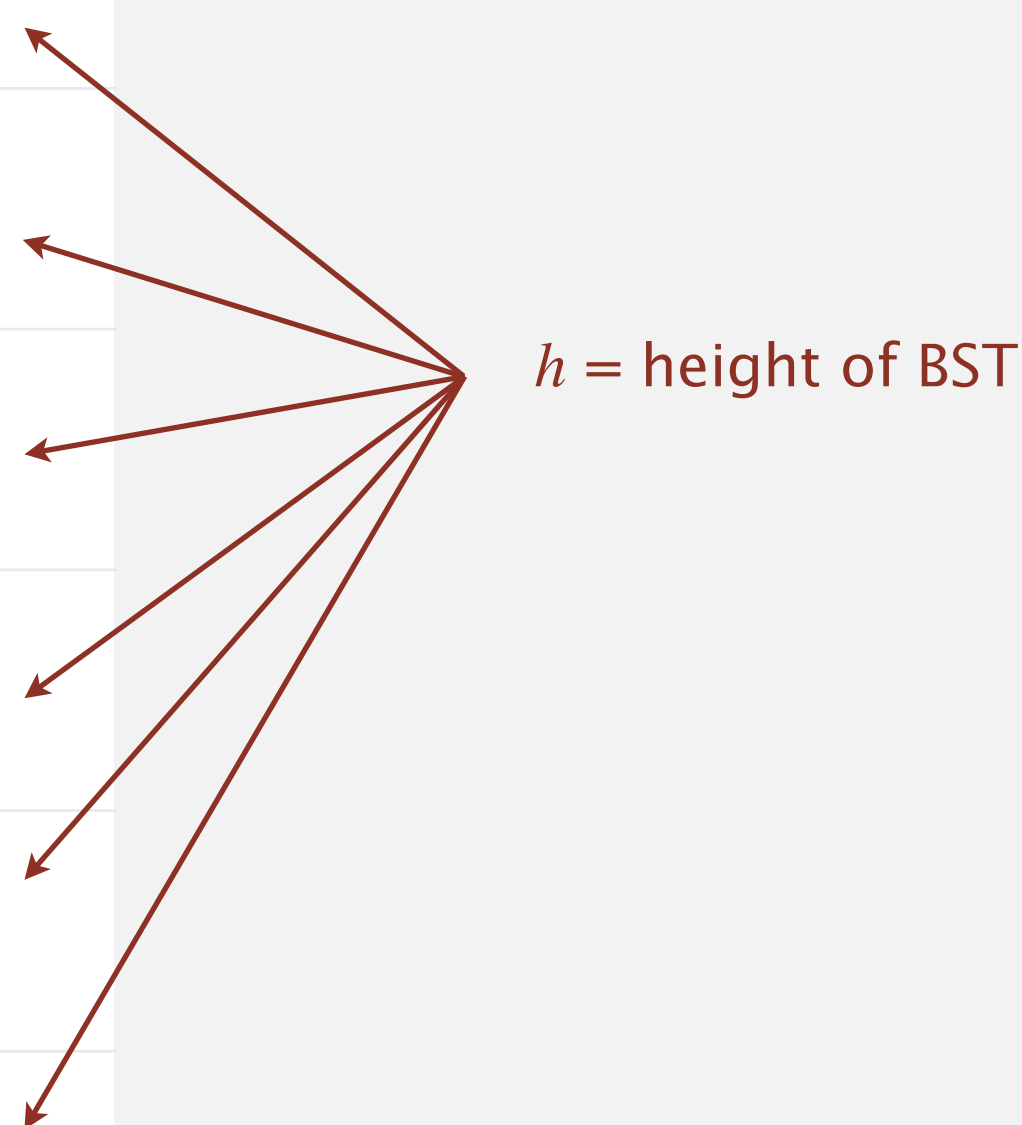
key in BST

key in node x is a candidate for floor
(floor can't be in left subtree of x)

key in node x is better candidate than champ

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	n	$\log n$	h
insert	n	n	h
min / max	n	1	h
floor / ceiling	n	$\log n$	h
rank	n	$\log n$	h
select	n	1	h
ordered iteration	$n \log n$	n	n



$h = \text{height of BST}$

order of growth of worst-case running time of ordered symbol table operations

ST implementations: summary

implementation	worst case		ordered ops?	key interface
	search	insert		
sequential search (unordered list)	n	n		<code>equals()</code>
binary search (sorted array)	$\log n$	n	✓	<code>compareTo()</code>
BST	n	n	✓	<code>compareTo()</code>
red-black BST	<div>log n</div>	<div>log n</div>	✓	<code>compareTo()</code>

next lecture: BST whose height is guarantee to be $\Theta(\log n)$

